Procedia Computer Science

Volume 51, 2015, Pages 110–119

ICCS 2015 International Conference On Computational Science

Improving OpenCL programmability with the Heterogeneous Programming Library

Moisés Viñas[1], Basilio B. Fraguela[1], Zeki Bozkus[2], and Diego Andrade[1]

[1] Universidade da Coruña, A Coruña, Spain
{moises.vinas, basilio.fraguela, diego.andrade}@udc.es
[2] Kadir Has Üniversitesi, Istanbul, Turkey
zeki.bozkus@khas.edu.tr

Abstract

The use of heterogeneous devices is becoming increasingly widespread. Their main drawback is their low programmability due to the large amount of details that must be handled. Another important problem is the reduced code portability, as most of the tools to program them are vendor or device-specific. The exception to this observation is OpenCL, which largely suffers from the reduced programmability problem mentioned, particularly in the host side. The Heterogeneous Programming Library (HPL) is a recent proposal to improve this situation, as it couples portability with good programmability. While the HPL kernels must be written in a language embedded in C++, users may prefer to use OpenCL kernels for several reasons such as their growing availability or a faster development from existing codes. In this paper we extend HPL to support the execution of native OpenCL kernels and we evaluate the resulting solution in terms of performance and programmability, achieving very good results.

Keywords: programmability, heterogeneity, portability, libraries, OpenCL

1 Introduction

The usage of accelerators has exploded in the past years. A crucial weak point of these systems is that they require much more programming effort than traditional CPUs, as they have separate memories that require additional buffers and memory transfers, special ways to launch code for execution, and need the specification of details that do not exist in CPUs. Another problem is that most of the tools to program these systems are specific to a family of devices [1][12][21], which severely restricts portability and may result in the rapid obsolescence of the applications built on them. OpenCL [14] is an answer to this latter problem that is gaining growing acceptance. Unfortunately, it is one of the environments that require more programming effort, particularly in the host side of the application [20], due to the low level of its API, even if we work in a object-oriented language such as C++.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.208

As a result of this situation, there have been several proposals to facilitate the usage of OpenCL in applications. Some of them are based on skeletons [6][25], so they are restricted to some computational patterns. Others have taken a more general approach [17][28], but they still leave users in charge of some tedious tasks or suffer from important restrictions. Another proposal is the Heterogeneous Programming Library (HPL) [26], which completely automates and hides all the management associated to OpenCL. HPL requires that the code portions to run in the accelerators are written in the language embedded in C++ that it provides. Nevertheless, users may prefer or even require to write their kernels in native OpenCL C for many reasons. For example, they may want to develop or prototype their kernels in OpenCL C so they can later integrate them in another project without adding HPL as another requirement for the project. Programmers may also want to take advantage of OpenCL C kernels provided by several projects [2][23]. Also, users may need to use native OpenCL C kernels because they want to use some of the automatic tuning tools available for them [7]. Finally, porting an existing application by placing the existing code into an OpenCL C kernel with a few minor adjustments, such as encapsulating it in a function, and adding some function calls to obtain the thread identifier, may require less effort than rewriting it with the HPL embedded language.

This paper extends HPL with a very convenient mechanism that allows it to use native OpenCL C kernels. These kernels can be freely mixed with kernels written in the HPL embedded language, and they enjoy the same benefits of total automation of the compilation process, buffer creation, data transfers, synchronizations, etc. The evaluation shows that the overhead of HPL over OpenCL is negligible, while the programmability improvement is remarkable.

The rest of this paper is organized as follows. Section 2 describes the Heterogeneous Programming Library. This is followed by the explanation of the new extensions in Section 3, a review of related work in Section 4, and our evaluation in Section 5. Finally, our conclusions are found in Section 6.

2 The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL) [26], available at `http://hpl.des.udc.es`, has a programming model that is very similar to that of CUDA [21] and OpenCL [14]. This way, the system where the application runs consists of a host with a general-purpose CPU in which the main application runs, and a series of devices connected to it, each one of them with its own processor(s) and memory. The processor(s) of each device must run the same code (in SPMD), and they can only access the device memory.

The portions of the application that run in the devices, called kernels, take the form of functions that can only operate on their arguments. Kernels are launched to execution specifying an n-dimensional space called global domain, where $1 \leq n \leq 3$, which indicates how many threads must run the kernel in parallel. Optionally, the threads can be grouped in subsets so that the threads in the same group can synchronize by means of barriers and share a fast scratchpad memory called local memory. The number of threads in each group is defined by a space of the same dimensions as the global domain, called local domain. Besides the local memory, the devices have a global memory that all the threads can access, and where the inputs and final outputs are stored. There is also a constant memory that the threads can read, but not modify, as well as a private memory that is exclusive of each thread.

HPL kernels are written in a C-like language embedded in C++ provided by the library with two characteristics. One is that C control constructs must be written finished by an underscore, the arguments of `for_` being separated by commas instead of semicolons. The second is that all the variables must have type `Array<`*type, ndim [, memoryFlag]*`>`, which represents an *ndim*-

```
1  void mxProduct(Array<float,2> c, Array<float,2> a, Array<float,2> b, Int p)
2  { Int i;
3
4    for_(i = 0, i < p, i++)
5        c[idx][idy] += a[idx][i] + b[i][idy];
6  }
7  ...
8  float cmatrix[M][N];
9  Array<float,2> c(M, N, cmatrix), a(M, P), b(P, N);
10
11 eval(mxProduct)(c, a, b, P);
```

Figure 1: Naïve matrix product in HPL

dimensional array of elements of the C++ type *type*, or a scalar for *ndim=0*. The optional *memoryFlag* allows to specify one of the kinds of memory available in the device (`Global`, `Local`, `Constant` and `Private`). The latter is the default for variables declared inside kernels. Similarly, non-scalars in the list of arguments of the kernel are assumed by default to be located in the global memory. HPL provides convenient data types to define scalars, characterized by an initial uppercase letter (`Float`, `Uint`, ...). Also, an analogous notation can be used to define vector types that are useful for SIMD computations (`Int8`, `Float4`, ...).

Figure 1 illustrates HPL with a program to perform a naïve matrix product `c=c+a×b`. The HPL predefined variables `idx` and `idy` identify each thread in the first and the second dimensions of the global domain. This way, in the kernel in lines 1-6 thread (`idx`, `idy`) computes `c[idx][idy]`. Line 11 illustrates how kernels are invoked in the host code. Namely, the syntax `eval(`*f*`)(`*arg1*, *arg2*, ...`)` where *f* is the kernel function, is used. While scalars of the standard C/C++ types are directly supported as kernel arguments in the host (but not in the kernel code), array arguments must also be declared in the host code with the `Array` type. Lines 8-9 show that these host-side `Array`s can be built in two ways. In particular, while the constructor of an `Array` always requires the size of each one of the dimensions, `Array`s defined in the host allow optionally as final argument a pointer to an allocated memory region that should be large enough to hold the data represented by the `Array`. If such pointer is not provided, HPL takes care of allocating and deallocating the memory needed as the object is built and destroyed, respectively. As for the number of threads to use, by default the dimensions of the global domain correspond to the dimensions of the first argument, while the local domain sizes are chosen by HPL, which suits our example. A number of modifiers to `eval` are supported, which allow to adjust these dimensions as well as to choose the device in which the kernel is to be executed. This way, `eval(f).device(d).global(80,60).local(20,30)(a,b)` requests the execution of kernel `f` in the device `d` (which is a handle of a type `Device`, provided by HPL) on the arguments `a` and `b` using a global domain of 80×60 threads divided in groups of 20×30 threads.

HPL must create buffers for the arrays that are not yet allocated in the target device and transfer the inputs from the host before a kernel can begin its execution. During the generation of the backend code for each kernel HPL identifies which are its input, output and both input and output arrays. In addition, the accesses to the `Array`s in the host code keep track of whether they are being read or written. The combination of these mechanisms allow the library to know where is the current correct version of every array and which are the arrays that need to be transferred when a kernel execution is requested or an array is accessed in the host, without

any user intervention. The transfers follow a lazy copying policy that minimizes the number of transfers, so that only when an access to a piece of data that is not available in a memory (either in the host or in any device) is requested, a transfer from the memory with the current version is performed.

In some situations higher performance can be achieved if the automatic management is avoided. Namely, the accesses in the host to `Array`s incur in a non-negligible overhead, as the consistency checks are performed in every single access. In fact these accesses are performed using parenthesis instead of square brackets[1] in order to visualize the extra cost incurred in the indexing of user-defined datatypes [9]. HPL provides a mechanism to avoid these overheads by means of the `data` method of its `Array`s. This method receives a flag with which the programmer can specify whether the array is going to be read, written or both, so that the library can perform its bookkeeping, and it returns a raw pointer to the contents to the `Array` that allows to directly access them.

Another useful feature of HPL is that kernel runs are asynchronous with respect to the host. This way, the host does not wait for the evaluation of a kernel to finish before proceeding with the execution of the main program. Rather, the host continues running its program in parallel with the kernel(s) execution(s) until a data dependency forces it to wait for an array to be generated by a kernel. This happens when an array that is written by the kernel is either accessed in the host or is part of the list of arguments of a kernel execution in another device. Nevertheless, if the new kernel execution takes place in the same device, the host simply issues the kernel execution request to the device, but it does not wait for the previous execution to finish, as each device runs its kernels in order.

Finally, HPL supports multiple devices [27] and provides other minor advantages such as a simple and powerful profiling system, or a structured error reporting system based on exceptions that can be caught and inspected using the standard mechanisms in C++.

3 Support for native OpenCL C kernels

While the semantics of the HPL embedded language are identical to those of C and its syntax is analogous, users may prefer or need to use native kernels written in OpenCL C for several reasons, the most important one being that this favors code reuse. We have extended HPL with a convenient interface that requires minimum effort while providing much flexibility. Our proposal requires defining a kernel handle that takes the form of a regular C++ function, and associating it to the native kernel code. After that point, the native kernel can be invoked using regular `eval` invocations on the kernel handle function. These invocations have exactly the same structure and arguments as those of the kernels written in the HPL embedded language, and they also fully automate the buffer creation, data transfer, kernel compilation, etc. that largely complicate OpenCL host codes.

A kernel handle is a regular C++ function with return type `void` (just as all kernels must be), and only its list of arguments matters. In fact its body will never be executed, so it is sensible to leave it empty. The arguments of the handle are associated one by one to the arguments of the kernel that will be associated to it. Namely, each kernel handle function argument must have the HPL type associated to the corresponding OpenCL C native type. This way, OpenCL C pointers of type `T *` will be associated to an `Array<T, n>` where `n` should be the number of dimensions of the underlying array for documentation purposes, although for correct execution it suffices that its value is 1 or greater. By default HPL arrays are allocated

[1] The accesses in the kernels use square brackets, as shown in Fig. 1, because kernels are compiled at runtime into a binary, thus having no overheads during their execution.

in the global memory of the device, so this suffices for OpenCL C pointers with the modifier `__global`. If an input is expected from `__local` or `__constant` memory, then `Array<T, n, Local>` or `Array<T, n, Constant>` must be used, respectively. As for scalars of type `T`, we can use an `Array<T, 0>` or the corresponding convenience type provided by HPL (`Int`, `Double`, ...).

While following these rules suffices for a correct execution, a kernel function handle defined with these arguments may incur in large overheads. The reason is that by default HPL assumes that the non-scalar arguments are both inputs and outputs of the associated kernel. This guarantees a correct execution, but it results in transfers between the host and the device that are unnecessary if some of those arguments are only inputs or only outputs. Our extension allows to label whether an array is an input, an output or both, so that HPL can minimize the number of transfers and follow exactly the same policies as with the kernels defined with its embedded language. The labeling consists in using the data types `In<Array<...>>`, `Out<Array<...>>` and `InOut<Array<...>>` in the list of arguments of the kernel handle function, respectively.

Once the kernel handle function has been defined, it must be associated to the native OpenCL C kernel code. This is achieved by means of a single invocation to the function `nativeHandle(handle, kernelName, kernelCode)`, whose arguments are the handle, a string with the name of the kernel it is associated to, and finally a string with the kernel OpenCL C code. The string may also contain other code such as helper functions, macros, etc. It helps programmability that HPL stores these strings in a common container, so that if subsequent kernels need to reuse previously defined items, they need not, and in fact should not, be repeated in the string of these new kernels. Also, it is very common that OpenCL kernels are stored in separate files, as it is easier to work on them there than in strings inserted in the host application and it allows to use them in different programs. The price to pay for this is that the application must include code to open these files and load the kernels from them, thus increasing the programmer effort. Our `nativeHandle` function further improves the programmability of OpenCL by allowing its third argument to be a file name. This situation is automatically detected by `nativeHandle`, which then reads the code from the given file. All the information related to the function is stored in a HPL internal structure that is indexed by the handle. The code is only compiled on demand, the first time the user requests its execution. The generated binary is stored in an internal cache from which it can be reused, so that compilation only takes place once. Altogether, `nativeHandle` replaces the IR generation stage explained in [26], being the compilation stage identical to that of the HPL language kernels. Finally, HPL also offers a helper macro called `TOSTRING` that turns its argument into a C-style string, avoiding both the quotes and per-line string continuation characters otherwise required.

The simple matrix product developed using the HPL embedded language shown in Fig. 1 has been transformed to use a native OpenCL C kernel in Fig. 2. The OpenCL kernel, called `mxmul_simple` is stored in a regular C-style string called `kernel_code`, and it is associated to the handle function `matmul`. Notice that since `eval` requires its arguments to be `Array`s, the kernel arguments are defined with this type in the host. Let us remember that it is possible to define them so that they use the data of a preexisting data structure, which facilitates the interface with external code. This strategy has been followed in this example with the `Array c`, which uses in the host the storage of the regular C-style matrix `cmatrix`.

4 Related work

There has been much research on the improvement of the programmability of heterogeneous devices. Some proposals identify important functions or patterns of computation, and provide

```
1  const char * const kernel_code = TOSTRING(
2    __kernel void mxmul_simple(__global float *c, const __global float *a, const __global float *b, int n)
3    { ... /* regular OpenCL C code goes here */ } );
4
5  void matmul(Array<float, 2> c, In<Array<float, 2>> a, In<Array<float, 2>> b, Int n) { }
6  ...
7  float cmatrix[M][N];
8  Array<float,2> c(M, N, cmatrix), a(M, P), b(P, N);
9  ...
10 nativeHandle(matmul, "mxmul_simple", kernel_code);
11 eval(matmul)(c, a, b, P);
```

Figure 2: Matrix product using native OpenCL C kernel with HPL

solutions restricted to them. This is the case of libraries of common operations [23][2], algorithmic skeletons [6][25] and languages for the representation of certain parallel patterns [5]. Some approaches combine several of these features. For example, [3, 16] provide both predefined functions and tools for the easy execution of custom kernels under strong restrictions, as they only support one-to-one computations and reductions.

Other works provide a more widely applicable solution by means of compiler directives [8][4][18][11][22]. This approach requires specific compilers and usually provides users little or no capability to control the result, which strongly depends on the capabilities of the compiler. Relatedly, these tools usually lack a clear performance model. These problems are even more important when we consider accelerators. The reasons are the large number of characteristics that can be managed, which leads to a much wider variety of potential implementations for a given algorithm than regular CPUs, and the high sensitivity of the performance of these devices with respect to the implementation decisions taken.

A proposal that also requires specific compilers but provides better control is [15]. It is more verbose than HPL because it presents more concepts to be managed by the programmer (accessors, queues, ...) and the usage of native OpenCL kernels, which is the focus of this paper, requires providing them as compiled OpenCL objects. In addition, the only currently publicly available implementation [13] is a mock-up that supports neither OpenCL nor accelerators.

The other family of proposals that enjoy the widest scope of application are libraries that improve the usability of the most common APIs, OpenCL in particular. These libraries[2] [23][17][28] require the kernels to be written using the native API, focusing on the automation of the tasks performed in the host code. A notable exception is HPL [26], which provides an embedded language that is translated into OpenCL at runtime. This latter strategy facilitates the integration of the kernels with the host application as well as the exploitation of run-time code generation.

The native OpenCL C kernels support in HPL proposed in this paper has several advantages and provides a higher-level view with respect to the related proposals we know of. This way, it is the only one that provides arrays that are seen as a single coherent object across the system, as the other solutions rely on a host-side representation of the array together with per-device buffers. While it is possible to avoid the host side representation for the buffers in [23][17] because they provide random element-wise accesses, each one of such accesses involves a transfer between the host and the device, and due to the enormous overhead, this is only very seldom

[2]We found other projects that are unsupported and/or miss academic references and that present the same characteristics as the ones discussed in this section, so we skip them for space reasons

Table 1: Benchmarks characteristics.

Benchmark	SLOCs host	Effort host	SLOCs kernels	Number of kernels	Number of arrays
FT	641	6118988	567	8	8
IS	394	2705245	571	11	12
EP	163	469038	238	1	2
ShaWa	186	893085	343	3	6

a reasonable solution. In addition, these buffers are not kept automatically coherent with their host image or with the buffers that represent the same data structure in other devices. Rather, they must be explicitly read or written. This makes sense because these proposals do not provide a mechanism to label which are the inputs and the outputs of each kernel, so their runtime cannot automate the transfers. For similar reasons, it is impossible for them to automatically enforce data dependencies between kernels run in different devices, or between kernel executions and arrays accesses in the host, unless by considering the most conservative, and therefore suboptimal, assumptions. Regarding devices, [17] only supports a single device, while [23][28] are based on the idea of selecting a current device, and then operating on it, including the explicit addition of each program to use to each device. This process is totally hidden in HPL [27], whose syntax for device selection is nicer and better supports multithreaded applications, as the current device approach requires critical sections when threads may operate on different devices. Also, [17] does not allow to define auxiliary functions, but only kernels, while [23][28] do not support local or constant memory arrays in the arguments.

5 Evaluation

This section measures the impact on productivity and performance of the usage of OpenCL kernels on top of HPL instead of the native OpenCL API. The evaluation is based on three codes of the SNU NPB suite [24] (FT, IS and EP) and a shallow water simulator developed in [19]. We ported the codes from C to C++, so that our baselines use the more succinct C++ OpenCL host API, which exploits all the advantages of this language such as its object orientation. This way the language characteristics play a neutral role in the comparison. We also encapsulated the initialization of OpenCL (platform and device selection, creation of context and command queue, loading and compilation of kernels) in routines that can be used across most applications and replaced these tasks with invocations to these common routines, so that they are not part of the evaluation. As a result our baseline corresponds to the bare minimum amount of code that a user has to write for these applications when using the OpenCL host C++ API.

Table 1 summarizes the most relevant characteristics of the baseline benchmarks with respect to the productivity evaluation. For each benchmark the number of source lines of code (SLOCs) excluding comments and empty lines for the host side code, the programming effort [10] of the host side code, the SLOCs of its kernels, the number of kernels and the number of arrays found in the arguments of the invocations of those kernels are listed. The programming effort is an estimation of the cost of the development of a code by means of a reasoned formula that is a function of the number of unique operands, unique operators, total operands and total operators found in the code. For this, the metric regards as operands the constants and identifiers, while the symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. We think that the programming effort is a fairer measurement of the productivity

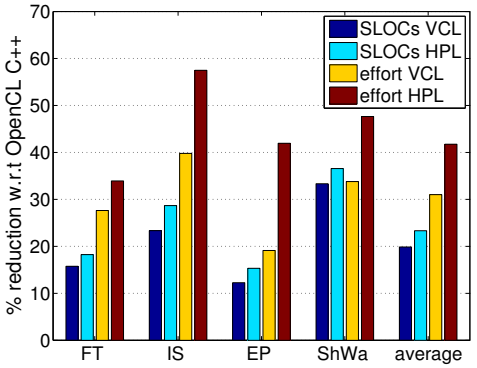

Figure 3: Productivity improvement in ViennaCL and HPL with respect to the baseline

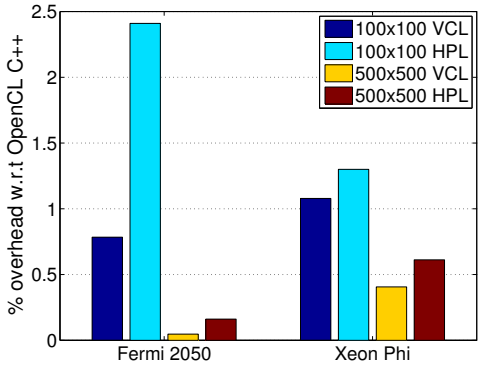

Figure 4: Overhead of ViennaCL and HPL in ShaWa with respect to the baseline

than SLOCs, as we all know that lines of code can widely vary in length and complexity. The SLOCs of the kernels are only given for informative purposes, as the changes only affect the host side of the applications. Finally, the number of kernels and related host-side arrays are relevant to interpret the productivity results, since once the usual initialization tasks required by OpenCL have been reduced to the minimum expression in the host code, the OpenCL related activities left basically focus on the creation of buffers, parameterization and execution of kernels, and the transfers between the device and the host.

In order to better gauge the advantages of HPL our evaluation includes ViennaCL [23] because it is one of the best alternatives for improving the usability of OpenCL in C++ (see Section 4) and it is a live and well supported project. We had to make some adjustments in applications with kernels that required local memory arrays and OpenCL vector types in their arguments to adapt them to ViennaCL because it does not support these possibilities.

Figure 3 shows the reduction of the SLOCs and the programming effort of the host side of our baseline applications when they are written with ViennaCL and HPL. The last group of columns represents the average reduction. Even when our baseline enjoys the C++ OpenCL API and the common boilerplate code required by OpenCL has been factored out, ViennaCL and HPL still provide average noticeable reductions of the programming cost metrics between 20% and 42%. Interestingly, the effort, which is more proportional to the actual programming cost than SLOCs, is the metric that obtains the largest reductions in all the benchmarks. Finally, HPL reduces SLOCs and particularly effort stronger than ViennaCL thanks to the advantages discussed in Section 4. It must be mentioned that the kernels were included in the host as a string. If they had been loaded from files, the automatic file loading feature of HPL would have allowed it to further improve programmability with respect to the other alternatives.

We also measured the overhead of ViennaCL and HPL with respect to the OpenCL C++ API in an NVIDIA Tesla Fermi 2050 with 3GB whose host has a Intel Xeon X5650 (6 cores) at 2,67GHz and 12GB RAM, and an Intel Xeon Phi with 60 cores at 1.056 GHz and 8GB with a host with 2 Intel Xeon CPU E5-2660 (8 cores per CPU) at 2.20GHz with 64GB RAM. The compiler was g++ 4.7.2 with optimization level `-O3`. The ViennaCL and HPL runtimes use the same strategies and functions as our optimized OpenCL baselines for data and kernel management, thus any time difference is related to the overheads of these libraries. We run very small tests, with class S for FT, IS and EP, and a 100×100 mesh for ShaWa to measure the overhead in the worst conditions, i.e., when the portion of the runtime associated to the

OpenCL management (not the kernel runs or host computations) is maximal. We also run more representative tests using FT class B, IS class C, EP class C and ShaWa on a mesh of 500×500 cells. The three versions achieved the same performance in both platforms for every benchmark except ShaWa. The reason is the very large number of kernel runs of this code, 492729 for the small test and 2517711 for the medium one, which allows to accumulate some overhead, shown in Fig. 4. As expected the larger automation of HPL generates more overhead than ViennaCL. However this is still very small, and it only reaches 2.5% on a baseline execution of just 17.86 s. in the GPU. For the more representative 500×500 runs this overhead falls to a negligible 0.16% and 0.61% in the GPU and the Xeon Phi, respectively.

6 Conclusions

Accelerators are here to stay, one of their weakest points being the code portability and programmability. OpenCL solves the portability problem, incurring however in larger programmer effort than other alternatives, particularly in the host side of applications. In this paper we have extended HPL to support native OpenCL C kernels in order to reduce this cost. Even when our baselines were very streamlined codes that used the OpenCL C++ API, our proposal reduced the number of lines and the programmer effort by a notable 23% and 42%, respectively, while imposing a totally negligible overhead on performance. Also, its productivity metrics, and particularly the programming effort, were consistently better than those of a comparable powerful approach such as ViennaCL.

Acknowledgements

This work was supported by Xunta de Galicia (Consolidation Program of Competitive Reference Groups Ref. GRC2013/055) and the Spanish Ministry of Economy and Competitiveness (Ref. TIN2013-42148-P), both cofunded by FEDER funds of the European Union. Zeki Bozkus is funded by the Scientific and Technological Research Council of Turkey (TUBITAK; 112E191).

References

[1] AMD. AMD stream computing user guide, 2009. v1.4.0a.

[2] AMD and AccelerEyes. clmath. http://github.com/clMathLibraries.

[3] N. Bell and J. Hoberock. *GPU Computing Gems Jade Edition*, chapter 26. Morgan Kaufmann, 2011.

[4] S. Bihan, G.E. Moulard, R. Dolbeau, H. Calandra, and R. Abdelkhalek. Directive-based heterogeneous programming. a GPU-accelerated RTM use case. In *Proc. 7th Intl. Conf. on Computing, Communications and Control Technologies*, 2009.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. 16th ACM symp. on Principles and Practice of Parallel Programming*, PPoPP'11, pages 47–56, 2011.

[6] J. Enmyren and C.W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th intl. workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, 2010.

[7] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. OCLoptimizer: An iterative optimization tool for OpenCL. In *Proc. Intl. Conf. on Computational Science (ICCS 2013)*, pages 1322–1331, 2013.

[8] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM parallel intelligent memory system. *ACM SIGPLAN Not.*, 38(10):49–60, June 2003.

[9] B.B. Fraguela, G. Bikshandi, J. Guo, M.J. Garzarán, D. Padua, and C. von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465–484, September 2012.

[10] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[11] T.D. Han and T.S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 22:78–90, 2011.

[12] IBM. C/C++ language extensions for Cell Broadband Engine architecture, 2006.

[13] R. Keryell. triSYCL. http://github.com/amd/triSYCL, 2014.

[14] Khronos OpenCL Working Group. The OpenCL Specification. V.2.0, Nov 2013.

[15] Khronos OpenCL Working Group-SYCL subgroup. SYCL 1.2 Provisional Specification, Sep 2014.

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157 – 174, 2012.

[17] O.S. Lawlor. Embedding OpenCL in C++ for expressive GPU programming. In *Proc. 5th Intl. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC 2011)*, May 2011.

[18] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proc. of 2010 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.

[19] J. Lobeiras, M. Viñas, M. Amor, B.B. Fraguela, M. Arenaz, J.A. García, and M. Castro. Parallelization of shallow water simulations on current multi-threaded systems. *Intl. J. of High Performance Computing Applications*, 27(4):493–512, 2013.

[20] R. V. Nieuwpoort and J. W. Romein. Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming*, 39(1):88–114, 2011.

[21] Nvidia. *CUDA Compute Unified Device Architecture*. Nvidia, 2008.

[22] OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.0a, Aug 2013.

[23] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, GPUScA, pages 51–56, 2010.

[24] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. 2011 IEEE Intl. Symp. on Workload Characterization*, IISWC '11, pages 137–148, 2011.

[25] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. In *2011 IEEE Intl. Parallel and Distributed Processing Symp. Workshops and Phd Forum (IPDPSW)*, pages 1176 –1182, may 2011.

[26] M. Viñas, Z. Bozkus, and B.B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *J. of Parallel and Distributed Computing*, 73(12):1627–1638, December 2013.

[27] M. Viñas, Z. Bozkus, B.B. Fraguela, D. Andrade, and R. Doallo. Developing adaptive multi-device applications with the Heterogeneous Programming Library. *The Journal of Supercomputing*. in press.

[28] R. Weber and G.D. Peterson. Poster: Improved OpenCL programmability with clUtil. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, pages 1451–1451, 2012.